JPA
Java Persistance API
O/R Mapping

JEE Microservices

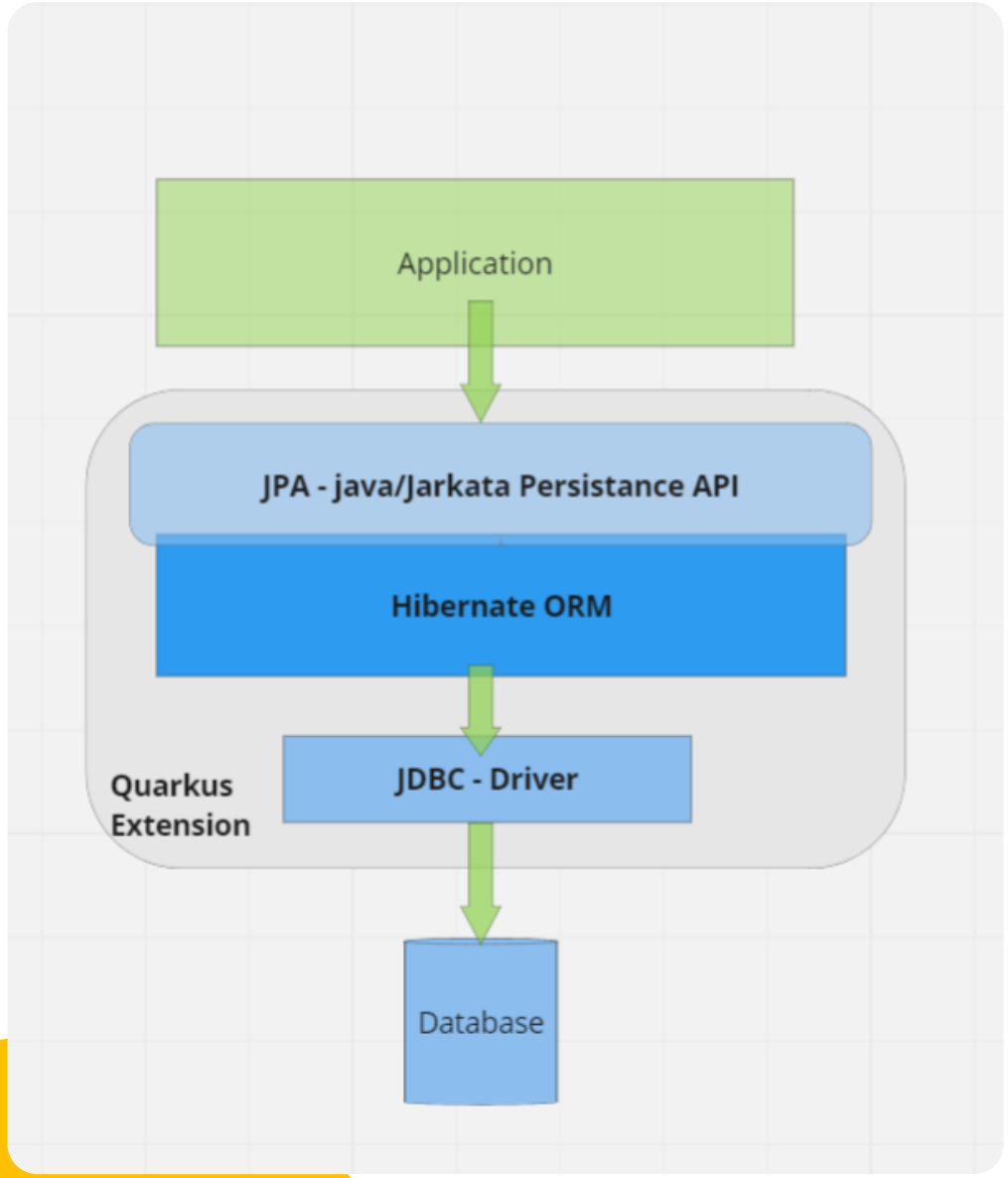
@ CGS IT – 2023

Version 1.0.5

QUARKUS

Content

- JPA Definitions
- JPA-Framework - Modules and APIS
- Entity Manager
- JTA - Transaction Management
- Entity Definitions and Annotations
- Mapping Examples
- JPQL
- Query API
- Quarkus Configuration



JPA Architecture

- Java Persistence API or now **Jakarta** Persistence API
- JPA provides Java developers with an object/relational mapping facility for managing relational data in Java applications.
- JPA itself is just a specification, not a product, it cannot perform persistence or anything else by itself.
- JPA is a set of interfaces and requires an implementation
- There are open-source and commercial JPA implementations to choose from and any Java EE 5 application server should provide support for its use
- JPA also requires a database to persist to

JPA – Framework Componentens

- The Java Persistence API
- Object/Relational Mapping Metadata
- The Query Language
- The Java Persistence Criteria API

JPA - API

- JPA specifies interfaces, Annotations and Patterns for OR-Mapping
- JPA enables a more Object-Oriented View do store objects to a relational database.
- Within the standard it also defines Entity Mapping Annotations
- It also defines a Query Language (JPQL) and
- a programatic Query API.

Modules and APIS used with JPA

- Java Bean Validation Framework
  - Integrates into JPA for additional bean validations during JPA operations
- JTA – Java Transaction API
  - Transaction Support via also Annotations for JPA
- Hibernate
  - Is often and used in Quarkus as an Implementation for JPA
- JDBC
  - JPA/Hibernate always connects to the database via a JDBC-Driver implemenation.

JPA – Entity Manager

- Main API is the EntityManger Interface
- API allows to manage Entities by providing all required
  - CRURD (create, read, update delete) methods
  - as well as search and query api utilites
- Getting an Instance for this API is simply done via @Inject

```
@Inject
EntityManager em;
```

JPA – Entity Manager – API

Method	Description
em.persist(Object entity)	Make an instance managed and persistent It will be attached to the EM Session and persisted during commit
Update ?	• An Entity is updated whenever you modify an Entity which is attached to a session, so loaded from the entity manager inside an active transaction • An Entity/Table can also be updated by directly using update statement queries
<T> T em.merge(T entity)	If an Entity was loaded via EM but the Entity Manager and Transaction is not active anymore, the Entity gets the status „DETACHED“. The Return value is the new attached merged Entity for the current EM Session
em.remove(Object entity)	Remove the entity instance from the Session, which leads to a db delete
em. find(Class<T> entityClass, Object primaryKey)	Find by primary key. Search for an entity of the specified class and primary key

JPA – JTA - Transactions

- The Java Transaction API specified by JSR 707
- The Api specifies Interfaces and Annotations to demarcate transaction boundaries
- Also defines an API to directly deal with the Transaction-Manager

JPA – JTA – Transactions and Scopes

Annotation	Description
@Transactional	Annoation to define TX Boundaries automatically Default Scope is : REQUIRED
@TransactionScoped	standard CDI scope to define bean instances whose lifecycle is scoped to the currently active JTA transaction
@Transactional(Transactional.TxType.***REQUIRED***)	Same as default. A Transaction is required. If no
Transactional.TxType. ***REQUIRES_NEW***	Always will create a new transaction for this method, even if another one is already active. If another TX is active the other TX will be suspended
Transactional.TxType.***SUPPORTS***	Participate in an existing TX or call the method without any additional TX. If you have read only methods todo you can use it
Transactional.TxType.***MANDATORY***	Mandatory is like Supports but if no TX is active a TransactionalException is thrown
Transactional.TxType.***NOT_SUPPORTED***	If the caller/client has a TX actve, it is suspended before processing continous
Transactional.TxType.***NEVER***	If the client is running in a TX context. A TransactionalException is thrown

JPA – JTA –Programatic TX-Management

- UserTransaction can also be manually injected
- And controlled manually also

```
public void insertChatMessageManualTX(ChatMessageEntity
newObject) throws SystemException, NotSupportedException {
   utx.begin();
   try {
      em.persist(newObject);
      utx.commit();
   } catch (Exception e) {
      utx.rollback();
      throw new RuntimeException("tx erorr ",e);
   }
}
```

As a rule of thumb use annotations
if no specific requirement needs to be fullfilled

Entities

- An entity represents a table in a relational database, and each
- entity instance corresponds to a row in that table
- An entity is a lightweight persistence domain object
- The primary programming artifact of an entity is the entity class

Entities - Basics

- The class must be annoyed with java.persistence.Entity
- annotation .
  - The class must not be declared final.
  - The class must implement the Serializable interface.
- Entities may extend both entity and non-entity classes, and nonentity classes may extend entity class.
- Persistent instance variable must be declared private, protected, or package-private.

Entity Mapping Annotations

Annotation	Description
@Entity	The POJO annotated with @Entity becomes a pesistant object.
@Table	Specifies a primary database table (name) for this entity
@SecondaryTable	Enables the storage into multiple tables for one Entity
@Embeddable	Annotates a a pojo used within Entites as an embedded class
@Inheritance @DiscriminatorValue @PrimaryKeyJoinColumn	Enables the Entity to use Java Inheritance and mapp it correctly to one or multipe database tables.

Entity Mapping Annotations - Fields

Annotation	Description
@Column	Specifies the mapped colum for a property in the Entity Class. Normally annotated on the property itself
@Id	Markes a property as a primary key column
@GeneratedValue	Used with ID columns to specify how the value is generated. @GeneratedValue(strategy=GenerationType. ***SEQUENCE***)
@Basic	The simplest type of mapping to a database column. Used especially with EAGER or LAZY loading flag
@Enumerated	Annotates the mapping for java enumerations by ordinal or String mapping. ***ORDINAL*** *is the default currently*
@Temporal	Used for date, time and timestamp mappings: @Temporal(DATE) protected java.util.Date endDate;
@Lob	for BLOBs /CLOB mappings
@Transient	A transient field is not persisted at all

Entity Mapping - Example

- Easy mapping for any Pojo into a Table
- If the @Table annotation is not used it defaults to the entity name
- NotBlank is already added as an example for bean valdation framwork usage

```
@Entity
@Table( name="chat_message")
public class ChatMessageEntity {

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id;

   @Column(name="user_name", length= 100, nullable = false)
   private String userName;

   @NotBlank(message="ChatRoom may not be blank")
   @Column(name="chat_room", length= 50, nullable = true)
   private String chatRoom;
  …
```

Entity Mapping Annotations - Relations

Annotation	Description
@OneToOne	1:1 relation to another Entity
@OneToMany	1:n relation to another entity used as annotation on the first „1“ side of the relation
@ManyToOne	Defines the N-side for a 1:n Relation (is a reference to the primary key of the target)
@ManyToMany	Used for n:m relations and used on both sides
@JoinColumn	Addional information for join columns

Entity Relation Mapping: Many to Many

- Both sides use @ManyToMany as Annotation
- The join table is specified on the owning side (customer)
- See ManyToMany Java Documentation for more examples

```
// In Customer class:

@ManyToMany
@JoinTable(name="CUST_PHONES")
public Set<PhoneNumber> getPhones() { return phones; }

// In PhoneNumber class:

@ManyToMany(mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

JPA - JPQL - Java/Jakarta Persistence Query Language

- Provides a SQL Like Query Language but against the mapped entity objects instead of the Database
- It provides select, Join, Where, order, group by and having support.
- See the specification and tuturials for more details

```
SELECT e from ChatMessageEntity e " +
    "WHERE e.chatRoom like :chatMessageLike " +
    "ORDER by e.creationTime DESC
```

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

See also:
https://en.wikipedia.org/wiki/Jakarta_Persistence_Query_Language
https://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html

JPA - JPQL - Query API

- Enables a programatic approach to queries by using a Criteria Builder
- and Expressions for the select and where conditions
- Is more save for refactoring

```
public Long countChatMessages() {

   CriteriaBuilder cb = em.getCriteriaBuilder();
   CriteriaQuery<Long> query = cb.createQuery(Long.class);
   Root<ChatMessageEntity> from =
query.from(ChatMessageEntity.class);

   Expression<Long> count = cb.count(from);
   query.select( count );

   // cq.where(/*your stuff*/);

   TypedQuery<Long> query1 = em.createQuery(query);

   Long singleResult = query1.getSingleResult();

   return singleResult;
}
```

Quarkus – JPA Configuration

- JPA Configuration is simplified via Quarkus to minimal settings
- The db-kind desides on a database type
- to use a database you need to import the driver maven dependency for the jdbc diver
- Datasource url and password

```
# datasource configuration
quarkus.datasource.db-kind = postgresql

quarkus.datasource.username = sc_admin
quarkus.datasource.password = sc_admin

quarkus.datasource.jdbc.url =
jdbc:postgresql://localhost:5432/simplechat

quarkus.hibernate-orm.database.generation=drop-and-create
# quarkus.hibernate-orm.database.generation=update

quarkus.hibernate-orm.log.sql=true
quarkus.hibernate-orm.log.bind-parameters=true
```

To see the generated SQL use ORM SQL logging parameters, which are also simplified a bit compared to hibernate direct logging configuration

Examples

Example JPQL Query

```
public Long countChatMessagsForRoom(String roomName) {

   // like query
   Query query = em.createQuery("select count(e) from
ChatMessageEntity e WHERE e.chatRoom like :chatRoomLike
");
   // :chatRoomLike wird durch den eigentlichen Wert ersetzt
   query.setParameter("chatRoomLike", "%room1%");

   // equals
   Query query2 = em.createQuery("select count(e) from
ChatMessageEntity e WHERE e.chatRoom = :chatRoomLike ");

   query2.setParameter("chatRoomLike", roomName);


   Long singleResult = (Long) query2.getSingleResult();
   return singleResult;
}
```

CGS ITSolutions

Example JPQL Query 2

```
@Transactional
public List<ChatMessageEntity>
findChatMessagesWithLikeNameAndOrdedByDate(String likeStatement)
{

   // suche alle chat messages and order by date descending

   Query query = em.createQuery(
         "SELECT e from ChatMessageEntity e " +
               "WHERE e.chatMessage like :chatMessageLike " +
               "ORDER by e.creationTime DESC" );

   if( !StringUtil.isNullOrEmpty(likeStatement)) {
      query.setParameter("chatMessageLike", likeStatement);
   } else {
      query.setParameter("chatMessageLike", "%");
   }

   List resultList = query.getResultList();

   return resultList; }
```

CGS ITSolutions

Examples

- Delete Query
- Delete without loading
- Delete from EM and DB for loaded Object

```
@Transactional(Transactional.TxType.REQUIRED)
public int deleteAllChatMessageEntity() {
   Query query = em.createQuery("DELETE FROM ChatMessageEntity");
   // returns the number of deleted entities
   int i = query.executeUpdate();
   return i;
}


@Transactional(Transactional.TxType.REQUIRED)
public void deleteWithoutLoading(Long idToDelete) {

   Query query = em.createQuery("delete from ChatMessageEntity where id =
:id");

   query.setParameter("id", idToDelete).executeUpdate();
}

/**
 * delete entity if object is already loaded
 * @param entity
 */
@Transactional(Transactional.TxType.REQUIRED)
public void deleteByObject(ChatMessageEntity entity) {

   em.remove(entity);

}
```

CGS ITSolutions

Example Critery Builder

```
@Transactional
public List<ChatMessageEntity>
findChatMessagesWithLikeNameAndOrdedByDateWithQueryBuilder(String
likeStatement ) {

   // "SELECT e from ChatMessageEntity e WHERE e.chatRoom like
:chatMessageLike ORDER by e.creationTime DESC"

   CriteriaBuilder cb = em.getCriteriaBuilder();
   CriteriaQuery<ChatMessageEntity> criteriaQuery =
cb.createQuery(ChatMessageEntity.class);
   Root<ChatMessageEntity> from = criteriaQuery.from(ChatMessageEntity.class);

   criteriaQuery.select( from );

   Predicate like = cb.like(from.get("chatMessage"), likeStatement);
   criteriaQuery.where( like );

   Order orderByCT = cb.desc(from.get("creationTime"));
   criteriaQuery.orderBy(orderByCT );

   Query query = em.createQuery(criteriaQuery);
   List<ChatMessageEntity> resultList = query.getResultList();
   return resultList;
}
```

CGS ITSolutions

Danke für Ihre Aufmerksamkeit

CGS ITSolutions